Interrupts and I/O I *

Jens Lechtenbörger

IT Systems, Summer Term 2024

This presentation is an introduction to interrupt and I/O processing, including a discussion of advantages and disadvantages of polling and interrupts.
This topic is covered in two presentations, of which this is the first one.

1 Introduction

Let us look at the core of our topic and its learning objectives, followed by a recap.

1.1 Today's Core Questions

- Recall keyboard handling in Hack.
  - You used a loop to wait for I/O; this is called **polling**.
- How can we improve I/O processing over polling?
  - Why keep CPU busy in loop when nothing happens?
  - With multitasking such time is **wasted** as other tasks could make better use of the CPU.
- Add **interrupts** as I/O notification mechanism.
  - How to organize I/O then?
  - How much overhead arises?
    * (Overhead: Additional indirect computation time; makes system less efficient.)
    * How to deal with "lots" of I/O events?

In this presentation, we revisit I/O processing in Hack, which is called polling. As observed already, polling is a waste of CPU time if no I/O events happen and other tasks could make better use of the CPU.
We look into interrupt processing as I/O notification mechanism, its overhead, and the question of how to deal with lots of I/O events.

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

1.2 Learning Objectives

- Explain techniques for I/O communication
  - Including sequencing of events under synchronous and asynchronous techniques (polling vs interrupts)
- Discuss dis/advantages of I/O communication techniques
- Explain (interrupt) livelock and mitigation via hybrid technique

Take some time to think about the learning objectives specified here.

1.3 A Note on Literature

- As an exception, this presentation is not based on (Hailperin 2019).
  - In (Hailperin 2019), Section 2.5 contains some introductory paragraphs on interrupts, while Section 7.3.1 starts with explanations on privilege levels and system calls.
- Chapter 1 of (Stallings 2014) as well of (Tanenbaum and Bos 2015) contain introductions on interrupts and I/O, while Section 5.1 of (Tanenbaum and Bos 2015) has additional explanations.

As an exception, this presentation is not based on our main textbook, but it collects information from several sources cited here, in addition to references on individual slides.

1.4 Retrieval Practice

- Before you continue, answer the following; ideally, without outside help.
  - How does the von Neumann architecture look like?
  - How does I/O processing work with Hack?
  - How to talk to an OS kernel?
  - How are programs, processes, and threads related?
  - What is multitasking? What are its advantages?

Please take a brief break and write down answers to these questions, without using previous class material.

1.4.1 Von Neumann and Hack Architecture

Take this quiz.

1.4.2 Computations

Take this quiz.

Agenda

- Part 1
  - Introduction
  - Hack vs Modern Computers
  - Polling
  - Interrupts
- Break for self-study
- Part 2
  - I/O Processing Variants
  - Outlook
  - Conclusions

The agenda for the current topic is as follows. After this introduction we contrast Hack with modern computers, highlighting important differences for the subsequent discussion. Then, we look at polling and interrupts for I/O processing.

Part 2 deals with I/O processing variants for improved latency. An outlook and conclusions end the presentation.

2 Hack vs Modern Computers

We start with a clarification of features of modern computers that we did not see for Hack.

2.1 Recall: Von Neumann Architecture

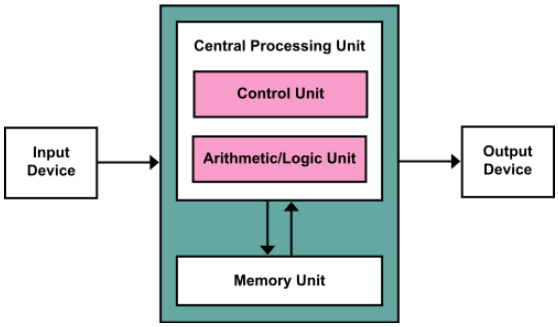

Figure 1: "von Neumann Architecture" by Kapooht under CC BY-SA 3.0; converted from Wikimedia Commons

Recall the von Neumann architecture. We consider computers with CPU, memory, and I/O devices.

2.2 Tabular Comparison

	Hack	Modern Computer (e.g., PC, smartphone)
Memory	• RAM for data, ROM for instructions • Physical RAM addresses • No secondary memory	• RAM for data and instructions • Physical and virtual addresses • Memory hierarchy (disks)
CPU	• Single core • Single mode of execution • Neither cache nor MMU	• Multi-core • Multiple protection domains • Caches and MMU
I/O	• No interrupts • Polling for I/O (recall keyboard)	• Interrupts, DMA • Different options for I/O
OS	• Language library • Single thread, no multitasking • No virtual memory	• Real OSs with system calls • Multitasking, scheduling • Virtual memory

This slide highlights key differences between the Hack platform and modern computers. We have already seen that data and instructions are kept separately in Hack, which is called the Harvard architecture, while modern computers keep data and instructions in RAM, which follows more closely the original von Neumann design.

You know that in Hack there are physical RAM addresses which are the bits that are fed into memory chips, while modern CPUs also support virtual addresses, which will be the topic of a separate lecture.

Also, modern computers come with a memory hierarchy, in particular including stable storage which is absent in the case of Hack.

Modern CPUs typically contain multiple cores; each of which you can think of as a full-fledged Hack CPU.

In addition, modern CPUs support multiple protection domains which for example allows protecting the operating system from interference of user applications.

Also, modern CPUs contain a memory management unit which is used in the context of virtual addresses and which therefore will also come back in later lectures.

With respect to I/O processing you already saw the programming technique called polling in the Hack platform to access the keyboard, which also exists for modern computers. However, modern computers also support interrupts and so-called direct memory access which requires additional hardware and is used for bulk transfer of data for example in the context of graphics cards. Thus, modern computers support different types of I/O processing and this will be the topic for today.

Finally, with respect to operating systems, Hack does not really have any operating system in a modern sense. If you take a look at the final book chapter, you'll see something called operating system, but it's really just a language library, and it does not support any of the major features of modern operating systems, such as multitasking or virtual memory.

2.3 CPUs in the Real World

- Known from Hack
  - Registers (addresses, data, control information)
  - Instruction execution cycle (e.g., fetch, decode, execute)
- Additionally
  - Caching
    - Cache = Small, fast memory between CPU and RAM
      - (Usually, multiple levels of caches; L1, L2, . . .)

    - Instructions and data must be in cache before CPU can access them

  Modern CPUs have several characteristics that are important in the contexts of OSs and performance. First, you know that according to the von Neumann architecture, the CPU fetches instructions for execution from RAM. As explained in the context of the memory hierarchy, CPUs are equipped with additional, fast but small memory chips called caches, which sit between RAM and processors, and data and instructions are loaded from RAM to a cache before the CPU accesses them. Subsequent accesses to cached data or instructions are much faster than original accesses in RAM, improving performance considerably.

    - Replacement policies when full (similar to those for memory management)
    - Overhead for context switches, **cache pollution**
      As caches are small, they come with replacement policies in situations where the cache is full and new contents need to replace old contents. We will see such policies in the context of memory management later on; if you are interested in details, maybe check out cache coherence protocols on Wikipedia.
      For now, note that computations are fast if lots of memory accesses can be served from caches instead of from RAM. However, you already learned that context switches happen between user space and kernel space, which as special case includes switching from user space to kernel space with scheduling of a different thread. In all those cases, the new execution context needs data and instructions that differ from the currently cached ones. Thus, context switches come with overhead in the form of loading new entries from RAM into caches. When returning to the old context, previously cached entries may have been replaced, which is also called cache pollution, and again requires slow accesses to RAM.

  - Special instructions and modes
    - Access to memory and devices in kernel mode: Subsequent slide
    - Enter idle state when nothing to do (save power)

  Furthermore, CPUs may support special instructions and execution modes to isolate kernel space from user space and user spaces from each other, for which basics are explained on the next slide. Besides, CPUs may come with special instructions to enter so called *idle states*, which can be executed by the OS to save power when no thread wants to execute any instruction. The link on this slide leads to details for Linux.

  - Interrupts: Later slides

    Importantly, CPUs have additional input pins or buses on which devices can trigger so-called interrupts to signal events that should be processed by the OS. Such interrupt processing is the topic of this presentation.

2.4 Privilege Levels/Rings/Modes

- Hierarchical **protection** domains of CPUs
  - Recall: **Kernel mode** vs. **user mode**
    - (See also Sec. 7.3.1 of (Hailperin 2019))
    - Governed by bit pattern in special register
- **Instruction set restricted** depending on mode/privilege level
  - Special registers protected
  - I/O, memory management protected

- OS starts in **kernel mode**
  - Applications run in **user mode**
  - Interrupts (system calls, traps, ...) lead into kernel mode

Modern processors support protection domains which means that pieces of code can be executed at different privilege levels.

For our purposes, it would be sufficient to consider just two privilege levels, the so-called kernel mode, which has full access to the underlying hardware, and the user mode, which is restricted and which is the mode in which ordinary applications execute. A bit pattern in a special register controls in which mode the processor is currently operating.

The key idea of protection domains is to restrict the instruction set which the CPU is currently able to execute. This restriction depends on the mode or privilege level at which the CPU executes instructions.

Maybe think about this question: What instructions might be allowed or forbidden in these modes?

As the current mode is recorded in a special register, that register of course needs to be protected. Otherwise, any code running in user mode could just access that register and elevate its own privilege level.

Thus, certain special registers are protected. In addition, I/O operations and memory management operations are also protected depending on the current privilege level.

The big picture for the use of different protection domains is as follows. The operating system starts in kernel mode with the highest privilege level. Consequently, it is allowed to do whatever it wants to do. It has full access over the underlying hardware. At some point in time, it starts the first user application, and it starts that application in user mode. Thus, the user application is restricted in terms of the instructions that it can execute and if it wants to perform I/O operations, for example, then it needs to perform a system call so that the operating system executes that operation in its kernel mode on behalf of the user operation.

In particular, system calls somehow need to switch the CPU from user mode into kernel mode. Similarly, interrupts also need to lead into kernel mode, and we will revisit that thought in the following.

As a side remark, note that this presentation is simplified. In particular, as remarked earlier and to be revisited in the context of virtualization, CPUs with virtualization support may also have something like a privilege level -1, where guest operating systems are running in a kernel mode but still do not have full control over the hardware, which is reserved to some supervisor component which runs at privilege level -1.

2.5 Big Picture

- I/O devices are components that **interact** with the OS
  - Some **receive requests** and **deliver results**
    - E.g., disk, printer, network card
  - Some **generate events** on their own
    - E.g., timer/clock, keyboard, network card

  The big picture of I/O processing is as follows.

  I/O devices are components that interact with the OS. Some I/O devices receive requests and deliver results, e.g., disk, printer, and network card.

  Some I/O devices can also generate events on their own, without any request, e.g., clocks used for timers, keyboard, and network card.

- Alternative types of I/O
  1. OS **polls** (continuously asks) for events/results
  2. Device triggers **interrupt** when event occurs or result is ready

  - (Historically, special CPU input pins/bits were used to signal interrupts; now, also existing buses are used, e.g., MSI)
  - CPU interrupts current computation and jumps into OS, which **handles** interrupt

Subsequently, we discuss two types of I/O processing, namely polling and interrupt processing.

With interrupt processing, I/O devices are active components that can notify the CPU when an I/O event occurs. When such a notification arises, the CPU interrupts its current computation and executes a special part of the OS, the so-called interrupt handler, which then deals with the I/O event. (As revisited later, interrupting the current computation requires some saving of state to resume the computation at a later point. Besides, interrupts can also be disabled when the CPU executes "important" tasks, which is beyond our topics.)

2.5.1 Types of I/O

- With polling, I/O is called **synchronous**
  - OS monitors I/O operation for completion
- With interrupts, I/O is called **asynchronous**
  - OS initiates I/O
  - I/O proceeds asynchronously, CPU free to perform other tasks
  - Device triggers interrupt when I/O operation completed
    - CPU interrupted, I/O result handled by OS

Regarding terminology, I/O with polling is called synchronous. Here, the OS monitors the I/O operation synchronously for its completion.

In contrast, I/O with interrupts is called asynchronous. Here, the OS triggers the start of an I/O operation. While the I/O operation proceeds asynchronously, the CPU is now free to perform other operations. Once the I/O operation completes, the device triggers an interrupt, which can then be handled by the OS.

3 Polling

Let us look at some details of polling.

3.1 I/O with Polling

I/O happens **synchronously** while OS polls for result

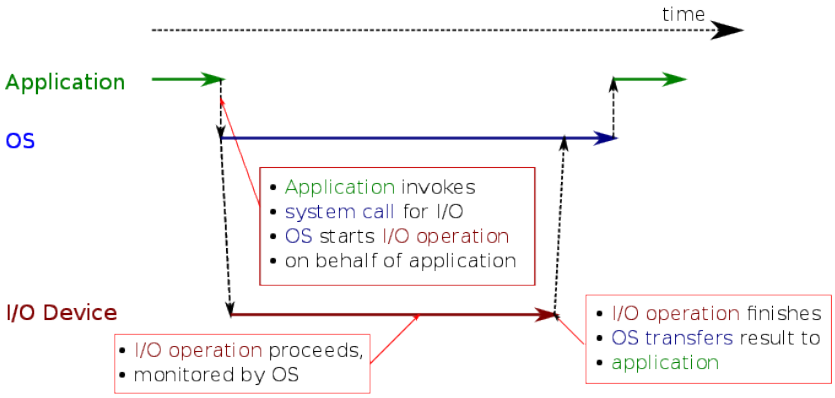

Figure 2: I/O with Polling

The key characteristic of I/O with polling is the following: While an I/O operation is ongoing, the CPU does nothing else but synchronously check whether the operation has successfully completed already.

In other words, we have the following key ingredients: first, an application that would like to perform some I/O operation, second the operating system which needs to perform the I/O operation on behalf of the application, third an I/O device, say a keyboard.

Recall that the application runs in user mode, while the operating system runs in kernel mode and is thus responsible for I/O operations.

After some normal processing, the application invokes a system call to have the operating system perform that I/O operation. Now, the OS starts that operation on behalf of the application at the device.

While the I/O operation proceeds, the OS continuously monitors its progress and waits for the result.

At some point in time the I/O operation finishes, the OS obtains the result, and transfers it to the application, which can then continue.

3.2 Polling Observations

- Advantages
  - Simple
  - Fast
    - Result processed as soon as it is available
      - No overhead (compared to interrupts as presented subsequently)
- Disadvantage
  - **Busy waiting** = waiting on CPU for event to occur
    - CPU time wasted if I/O is slow or infrequent
    - Bad idea if wait period is "long"

The question is, how good is polling as an I/O processing technique?

It turns out that this question is a hard one, and both parts of this presentation deal with that question. Let us take a look at advantages and disadvantages of polling.

The major advantage of polling is that it is simple to program. Think of a thread that would like to perform an I/O operation, for example reading a file from disk. Quite likely the thread invokes the I/O operation because it needs the file contents to proceed reasonably. In

this situation, the simplest approach is to issue that I/O request and then just "to poll", i.e., to check continuously whether the data has arrived already.

However, waiting for something that is happening externally looks like a waste of CPU time: The CPU could theoretically do something useful but all that it does is wait for I/O to happen.

An alternative would be to program the thread in such a way that it can do something else while disk data is being transferred. Apparently, that would make the program much more complex. Also, polling is faster than this alternative. If a thread actively waits for data from disk, then it can respond to that data as soon as it is available, no overhead involved at all. This point will become much clearer once we have taken a look at the overhead involved in interrupt processing.

Polling is a special instance of a technique called busy waiting. Busy waiting is also discussed in the book by Hailperin in the context of scheduling, which is the topic of a subsequent presentation.

3.2.1 Polling Example: Hack Keyboard

- Program waits for user to press key
  - Loop, repeatedly reading keyboard memory location until key pressed
  - Recall memory-mapped I/O
- CPU executes instructions all the time
- Even if Hack had OS with multitasking, nothing else could be executed
  - CPU time in loop is **wasted**

As an example for polling, recall that you programmed a loop to wait for Hack keyboard input.

4 Interrupts

Let us now turn to interrupt processing.

4.1 Fundamental Idea

Interrupts are similar to function calls in the sense that if a function gets called then code elsewhere in memory gets executed.

- **Interrupt**: Signal to CPU
  - Generated **externally** to CPU (signal on bus or separate pin)
    - CPU stops doing whatever it did
    - CPU jumps (resets program counter) to **interrupt handler** instead (details on following slides)

  Similarly, when an interrupt gets triggered, the CPU stops doing whatever it did so far. Instead, it jumps to an interrupt handler, which is part of the OS.
- If I/O devices generate **interrupts**, CPU does **not** need to wait for I/O completion
  - OS **initiates** I/O operation at device
    - CPU is free to do something else **asynchronously** during I/O execution

- At later point, I/O operation completes and **device triggers an interrupt**
  - OS interrupt handler acts accordingly

The major benefit of introducing interrupts is that the operating system no longer needs to actively wait for the completion of a potentially long-running I/O operation, but instead it just needs to initiate the operation. Afterwards, it is free to do something else asynchronously, while the I/O operation is ongoing.

When at some later point in time the I/O operation completes, the device triggers or raises an interrupt and the operating system's interrupt handler acts accordingly.

4.2 I/O with Interrupts – Overview

- **Asynchronous** processing of I/O
  - External notifications via interrupts

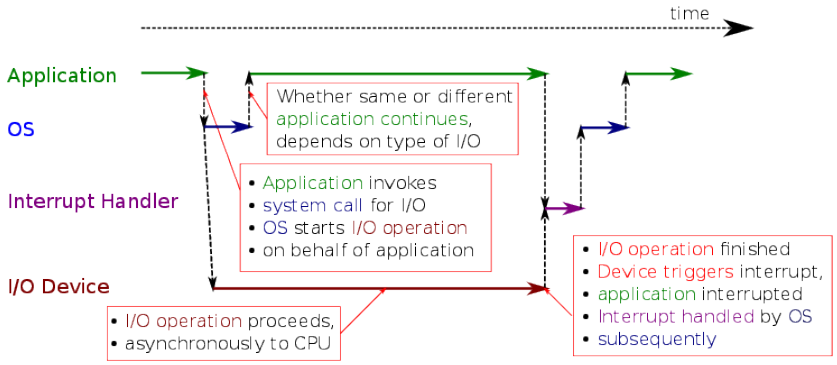

Figure 3: I/O with Interrupts

This slide shows the big picture of input/output processing using interrupts. I really recommend that you contrast this slide with the earlier one shown for I/O processing with polling.

Similarly to the polling case, we look at the application, the operating system, and the I/O device. In addition, we now take a look at the interrupt handler, which is the specific part of the operating system that is responsible for handling interrupts.

Exactly as in the case of polling, an application may at some point in time invoke a system call to perform an I/O operation, and the OS is responsible for starting the I/O operation on behalf of the application.

This time however, the OS only initiates the I/O operation and is then free to do something else while the I/O operation proceeds asynchronously.

After the OS has performed all housekeeping and management operations that it needs to perform, an application can continue running. Actually, whether the previously running application or a different application continues, depends on the type of I/O, and we will take a look at that separately.

When the I/O operation is finished, the corresponding device triggers an interrupt, and that interrupt then leads to execution of the interrupt handler of the OS, which takes care of necessary management operations.

Finally, the OS allows some application to continue.

4.2.1 Hypothetical Interrupt Example: Hack Keyboard (1/3)

- Suppose Hack had multitasking OS with threads and interrupts (which it does **not**)

  - Multiple programs could run concurrently in separate threads

- Again, wait for user to press key
  - This time, thread invokes (blocking) system call, asking OS for next pressed key
    * (Non-blocking system calls exist as well, discussed in part 2)
  - OS remembers to inform that thread about keyboard input, **blocks** it
    * OS takes that thread aside
    * OS dispatches different thread to execute on CPU

Consider a hypothetical example how interrupt handling could look like for keyboard handling with Hack.

Suppose Hack had a multitasking OS with threads and interrupts, which it does not. Then, multiple programs could run concurrently in separate threads.

If one of those threads needed keyboard input, it would invoke a blocking system call, asking the OS for the next pressed key.

In response, the OS would remember to inform that thread about keyboard input, and blocks it. This means that the OS takes that thread aside and dispatches a different thread to execute on the CPU.

4.2.2 Hypothetical Interrupt Example: Hack Keyboard (2/3)

- Eventually, key gets pressed
  - Keyboard triggers interrupt
    * CPU interrupts whatever it does, jumps to interrupt handler, which interacts with hardware to obtain value representing key
    * OS records that value as return value of system call
      · OS **unblocks** blocked thread
  - Scheduling decision by OS determines what (unblocked) thread to continue next
    * At some point in time, maybe right now, OS chooses thread that invoked the keyboard system call
      · Thread continues with return value from system call

Eventually, a key gets pressed, and the keyboard triggers an interrupt. Now, the CPU interrupts whatever it does, jumps to the interrupt handler, which interacts with hardware to obtain a value representing the pressed key. The OS records that value as return value of the system call and unblocks the blocked thread.

At some point in time, the OS performs scheduling to determine what (unblocked) thread to continue next. Eventually the thread that invoked the keyboard system call is scheduled. That thread can now continue with the return value from the system call, which indicates the pressed key.

4.2.3 Hypothetical Interrupt Example: Hack Keyboard (3/3)

- Notice: Latency (delay) between system call and processing of its return value
  - Latency between system call and key press
    * Cannot be avoided

        * In contrast to polling, with interrupts something useful can happen on the CPU during this period

          Notice that different types of latency, or delay, occur between the system call and the processing of its return value. First, it takes an unpredictable amount of time, before a key gets pressed. Clearly, this latency cannot be avoided. However, during that time, something useful can happen on the CPU if interrupts are used.

    - Between key press (interrupt) and return of key's value into thread
        * Processing of interrupt with **overhead**
            · Discussed subsequently
        * Resulting delay does not exist for polling

      Second, there is an additional latency between the points in time when the key is pressed and when the value for that key is returned to the thread. This latency arises because processing of interrupts comes with an overhead, which is discussed subsequently. This additional latency does not exist for polling.

4.3 Interrupts, Traps, Faults, Exceptions

- Interruption of ordinary CPU execution
- Hardware-specific, terminology not unified
- Classification
    - **Asynchronous**, triggered externally
        * **Hardware interrupt** (e.g., key pressed, data received)
            · **Timer** (clocks); basic mechanism in scheduling presentation
    - **Synchronous**, triggered internally
        * **Software interrupt**
            · Before execution of instruction, e.g., **page fault** as basic mechanism in virtual memory presentation
            · After execution of instruction (e.g., overflow)

Here, you see various terms related to interrupts, namely interrupts, traps, faults, exceptions. Terminology is not well-defined and varies from textbook to textbook and processor to processor. In all cases, we are talking about interruptions of the ordinary CPU execution.

An accepted classification of those interruptions is into the classes of asynchronous and synchronous interruptions. Note that in this context, the words "asynchronous" and "synchronous" are not directly related to synchronous and asynchronous I/O processing.

Asynchronous interruptions are those that are triggered externally to the CPU, for example hardware interrupts, i.e., if a key is pressed or if a disk has transferred a block of memory. In addition, clocks or timers can raise interrupts, which is useful to trigger interrupts periodically, which will be important for scheduling purposes later on.

In contrast, synchronous interruptions are those that are triggered internally to the CPU. One example are software interrupts, at which we will take a closer look in a minute. In addition, interruptions can occur either before or after the execution of an instruction. In particular, page faults will become important in the context of virtual memory management later on. In contrast, interruptions after instructions are not important for our purposes.

4.4 Interrupts and CPUs

- OS specifies a **handler** for each type of interrupt and exception
  - Handler = function
  - Type of interrupt determined by number
- E.g., x86 processors
  - Addresses of handlers stored by OS in in-memory table, the **Interrupt Descriptor Table** (IDT)
    - Synonym: Interrupt Vector (Table)
    - Each table entry points to one handler/function
  - CPU core contains **Interrupt Descriptor Table Register** (IDTR)
    - OS initializes IDTR with start address of IDT

This and the subsequent slide provide more details for interrupt processing on our CPUs. From previous explanations, it should already be clear that there are different types of interrupts and exceptions, for example timer interrupts, interrupts when keys are pressed, and also page faults.

To deal with this variety, the operating system specifies a handler for each type of interrupt. You should think of such a handler just as a function, and the type of the interrupt is identified by some number.

The operating system stores addresses of all these functions, all these handlers, in a table in memory, which is called interrupt descriptor table, IDT for short. Sometimes you also see the term interrupt vector table for that purpose. The idea is as follows: This table contains an entry for each handler that specifies where the handler resides in main memory. In addition, the CPU has a specific register that contains the start address of the IDT, namely the interrupt descriptor table register, IDTR for short. When the operating system starts, it creates the IDT in memory and places its start address into this special register.

4.5 Interrupt Handling

- Upon interrupt of type n:
  - A **context switch** takes place, and (in kernel mode) the CPU
    - saves state of current execution,
    - uses IDTR to access IDT,
    - looks up entry n in IDT, and invokes corresponding handler/function.
    - Afterwards, state is restored, previous execution continued.

      Whenever an interrupt occurs, a context switch takes place: The CPU gets interrupted in whatever it is currently doing. To resume that interrupted execution, it needs to save the state of the current execution. Then, it uses the IDTR to look up the start address of the IDT. Based on the type of interrupt that has just occurred it looks at the corresponding entry in the IDT, obtains the start address for that handler, and jumps to the handler at that location, which handles the interrupt.

      Once interrupt handling has finished, the CPU restores the saved state and resumes the previously interrupted execution.
- Context switch comes with **overhead**
  - Save/restore state
  - Cache pollution

        * Cache contents unlikely to be useful in new context
            · (In particular, this affects the so-called TLB, to be discussed in virtual memory presentation)
    - Maybe scheduling (later presentation)
        * With setup of new address space (virtual memory presentation)

Note that such a context switch always comes with some overhead. E.g., the CPU needs to save and restore the state; depending on what exactly happens in the course of interrupt handling, it may also be necessary to set up separate address spaces. Then, the execution of instructions elsewhere in memory leads to cache pollution: Cache entries that were previously useful, are no longer useful in the context of the interrupt handler, and therefore they may be evicted from the cache, to be replaced by interrupt handling specific data and instructions. Later on, when execution continues in the original context, its data and instructions are no longer present in the cache.

Similar effects happen for the so-called Translation Lookaside Buffer which deals with virtual address translation, which will come back in a later lecture. In addition, maybe scheduling is necessary to figure out what to do next.

All these steps take time, leading to overhead, which may slow down the system.

4.6 Aside: Interrupts for System Calls

- Recall: System calls = API provided by OS kernel
    - Implementation is OS and hardware specific
- Hardware specific methods to **enter kernel mode**
    - `int 0x80` (generic), `SYSCALL` (AMD), `SYSENTER` (Intel)
- Beyond class: Linux, Intel IA-32, `int 0x80`
    - **Software interrupt** via `int 0x80` leads into kernel mode
    - IDTR contains address of IDT
        * Entry `0x80` points to handler function
            · In the past `system_call`, since 2015 `entry_INT80_32`
        * Initialization during boot (`arch/x86/kernel/traps.c`)

A previous slide contained the term software interrupt as an example for a synchronous interruption of the CPU. Now let us take a look at how this kind of interrupt can be used to implement system calls.

Remember that system calls are the application programming interface provided by the OS kernel and that the OS runs in kernel mode, while user applications that invoke system calls run in user mode. Essentially, the OS needs to use some mechanism which switches the CPU from user mode to kernel mode. Different hardware specific machine instructions exist to enter the kernel mode. The slide names a couple of those.

For example, on a 32-bit Intel processor the instruction int 0x80 triggers a software interrupt, which switches the CPU into kernel mode, where the IDTR is used to look up the start address of the IDT as explained previously. The IDT in turn is then used to look up the start address of the specific function or handler to execute. In the case of Linux, you can read about that function at the URL linked here. Also, you can take a look at the source code file linked here to see all the details of initialization.

4.7 Self-Study Quiz

This task is available for self-study in Learnweb.

- Consider a networked machine that receives incoming messages. Each of those messages requires about 4 µs CPU time for processing. If interrupts are used, each interrupt introduces a delay of about 6 µs (caused by different types of overhead).
  - How many messages can be processed per second with polling, how many with interrupts?
  - How much time is wasted (waiting for messages to arrive) with polling in the worst case? How much spent on overhead processing with interrupts?

Please take a break for this self-study task.

Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1. https://gustavus.edu/mcs/max/os-book/.

Stallings, William. 2014. *Operating Systems – Internals and Design Principles*. 8th ed. Pearson.

Tanenbaum, Andrew S., and Herbert Bos. 2015. *Modern Operating Systems*. 4th ed. Pearson.

The bibliography contains references used in this presentation.

License Information